Süsteemprogrammeerimine keeles C

Loeng 14

Milles tehakse selgemaks ja loogilisemaks tehete järjekorda puutuv ja räägitakse veidi chroot süsteemikäsust, peatume ka sellel, mida teevad assert ja sellel mis probleeme võib esineda paroolidega. Lõpetuseks paar sõna keywordide kohta.

Eelmises osas(Lõimed ja fork())

fork() kopeerib kogu protsessi mälu

fork() ei kopeeri lõimesid

ei ole sünkroniseeritud

phtread_atfork(prepare, parent, child)
idee selles, et prepare() lukustab mutexid, parent() laseb lahti. child() initsialiseerib oma mutexid ise

Eelmises osas(Stream, lõimed ja fork())

Standardlibra kaitseb streamide listi oma varjatud mutexiga, sellega saab fork() hakkama

Üksikute streamide puhul peavad kõik streamid olema forkiva protsessi käes – flockfile() ja funlockfile()

pärast fork() käsku ei ole mutex enam jagatud ja hakkab ralli kahe protsessi vahel nagu võib arvata

Eelmises osas(Lõimepõhised muutujad)

Thread Specific Data poole pöördumiseks on TSD key
phtread_key_t tüübist

pthread_key_create (key, destr_function)
pthread_key_delete(key)
pthread_key_setspecific(key, pointer);
pthread_key_getspecific(key)

Socketite paar

Iseendaga suhtlemiseks saab süsteemilt tellida kaks socketit kahepoolseks suhtluseks

```
int filedes[2];
socketpair(AF_LOCAL, SOCK_STREAM, 0, filedes)
```

tagastatakse ilus kahesuunaline suhtluskomplekt

Eelmises osas (Signaalid uuesti)

Signaal: tarkvaraline katkestus programmile

Tekkimisel läheb ootele *(pending)*
Blokeeritud signaal ootab, muidu saadetakse protsessile

Saabumisel vaadatakse, mis teha
Valikud: ignoreerida, vaikimisi tegevus teha, lasta *handler* käima

Eelmises osas (Signaali haldamine)

Kaks põhistrateegiat:
- handler teeb vähe (1 globaalmuutuja mudimine)
- handler katkestab töö või teeb longjump()

Arvestada: signaalid võivad tulla millal iganes; teine handler võib käivituda esimese töö ajal

Eelmises osas (Handlerite tüübid)

Tagastuv handler: jätkab väljumisel poolelijäänud kohast; ei sobitu veaolukordadele, peab muutma globaalmuutujaid (deklareeri kui volatile)

Tööd lõpetav handler: valmistab ette programmi töö lõpu, võtab handleri vastava signaali puhul maha, ning raise() funktsiooniga tõstab sama signaali aktiivseks

Hüppega handler: jälgi, et kriitilised andmestruktuurid poleks poole kirjutamise peal (blokeeri signaalid kirjutamisel või initsialiseeri/korrigeeri pärast hüpet)

Eelmises osas (mitmekordne käivitatavus)

Handleris funktsioonide käivitamisel peab jälgima, et nad oleksid mitmekordselt käivitatavad (*reentrant*)

Mitmekordse käivitamise välistab globaalmuutuja kasutamine, staatilise muutuja kasutamine või mingi välise objekti (stdout) kasutamine.

Malloc ja free pole enamasti taaskäivitatavad, errno-d muutvad pole (seda saab parandada), mälu lugemine ja kirjutamine on oma olemuses ohutu - mõtle

Eelmises osas (Signaali blokeerimine)

Me ei taha kriitilistel hetkedel signaale saada.
Blokeerime ajutiselt mingi hulga signaale.

sigset_t
sigemtptyset(sigset); sigfullset(sigset);
sigaddset(sigset, signum); sigdelset(sigset, signum);

sigprocmask (how, setptr, oldsetptr); määrab protsessi signaalimaski; how – SIG_BLOCK, SIG_UNBLOCK, SIG_SET

Eelmises osas (Terminalist)

Kanooniline režiim – kasutaja muudab oma rida kuni vajutab ENTER, siis edastatakse programmile

Mittekanooniline režiim – kasutaja nupuvajutused lähevad programmile otse

termios.h sisaldab palju temaatilist värki

Eelmises osas (Curses)

Curses – hulk libras defineeritud funktsioone curses.h või ncurses.h

Võimaldab terminalis luua/kujundada aknaid, seda terminalitüübist sõltumatul viisil.

Võimas, aga seega keeruline

Eelmises osas (Debugimisest)

GDB kasutamiseks kompileeri -g võtmega
run – käivitab; bt – backtrace

Võib defineerida makro, mis programmis käitatava funktsiooni, rea jms välja trükib

backtrace_symbols() laseb parajasti kehtiva stacki välja.

Tänases osas

- Operaatorite järjekord
- chroot
- assert
- getpass
- crypt
- keywordid

Operaatorite järjekord (precedence)

```
>>    ()    []   ->   .
<<    !   ~   ++   -- + - * & (type) sizeof
>>    *   /   %
>>    +   -
>>    <<   >>
>>    <   <=   >   >=
>>    ==   !=
>>    &
>>    ^
>>    |
>>    &&
>>    ||
<<    ? :
<<    =   +=   -=   *=   /=   %=   &=   ^=   |=   <<=   >>=
>>    ,
```

Selgitus eelmisele

- << ja >> (paremalt vasakule ja vasakult paremale) näitavad mis suunas operaator lahendub/assotsieerub
  - sisuliselt see, et kummalt poolt tehte lahendamist alustatakse
- Samal real olevatel tehtel on sama prioriteet (lahendumise järjekord näitab millised ennem)
- Eri ridadel on erinev prioriteet, enne tehakse kõrgema prioriteediga tehted

Järjekord sammhaaval

- Tehete järjekorra äraõppimiseks on mõistlik tekitada süsteem.
  - Õnneks on järjekord üsna loogiline
- Tehetel on järjekord selleks, et programmeerijal oleks tarvis minimaalselt sulge panna.
- Eeldame, et meil on hunnik tehteid ja soov neid järjekorda seada. Kuidas käituda?
- Esiteks on lihtsam, kui asju grupeerime: otsustame, et ühesugused asjad on grupis

Matemaatikutelt laenatav

- Võtame skaala aluseks matemaatikute poolt antavad tehted: + - * /
- "Enne korrutan ja jagan, siis liidan ja lahutan"
  - 4 klassi matemaatikaõpik
- Juurde tuleb jäägiga jagamine %, mille kohta arvame, et tegu on korrutise ja jagamise tähtsusega tehtega
- Assotsieeruvus on vasakult paremale, seega tehakse tehted nagu oleme harjunud
  - 4 / 2 + 10 * 2 = (4 / 2) + (10 * 2)

Loogikutelt laenatav

- Laenata saame ka loogikutelt
  - AND tehe seob tugevamini kui OR (AND käitub väga 1 ja 0 omavahelise korrutamise moodi; OR on peaaegu, et liitmine)
- XOR tehte paneme JA ja VÕI vahele
  - meeldejätmist võib põhjendada kasvõi sellega, et see on natuke ebaloogiline (või sellega, et DNK mudelisse sobitub ta JA ja VÕI vahele)
- Lisaks otsustame, et bitthaaval tehtavad tehted toimuvad enne (sest keerulisem bitthaaval toimuv tehe on kindlasti olulisem, kui labane nulliga võrdlemine.)

Vahekokkuvõte

```
                (enne aritmeetikat tehtavad)

>>      *   /   %     (aritmeetikatehted)
>>      +   -
                (aritmeetika ja loogika vahel
                 olevad)

>>      &
>>      ^
>>      |           (loogikatehted)
>>      &&
>>      ||

                (pärast loogikat olevad)
```

Mis tuleks ära teha enne aritmeetikat?

- Kõik unaarsed (ühe argumendiga) tehted; paneme nad sama prioriteediga ritta.
- Assotsieeruvad paremalt vasakule
  - Märki näitavad "tehted" nagu + ja -
  - Aadressioperaator & ja mälu sisu operaator *
  - Suurendamine ja vähendamine ++ ja --
  - Bitthaaval eitus ~
  - Tavaline eitus !
  - sizeof ja *cast*imine (*tüübinimi*)

Mis on unaarsetest veel olulisem?

- Selles grupis on need tehted, mis võiks analoogselt unaarsetega toimuda enne liitmist ja jagamist, aga omakorda ka enne kui unaarseid tahame teha.
- Siia paneme sulud (), et sulud päriselt töötaksid
- Siia paneme nurksulud [], et massiivid näitaks enne nendega tehete tegemist õigesse kohta
- Siia paneme pointerviidatud->struktuurielemendi ja struktuuri.elemendi , et ka need enne kõiksugu märkide andmist olemas oleksid
  - Rohkem midagi olulisemat ei ole

Vahekokkuvõte 2

```
>>    ()   []  ->  .                          {eriti olulised}
<<    !  ~  ++  -- + - * & (type) sizeof  {unaarsed}
>>    *  /  %                                 {aritmeetika
>>    +  -                                    }



>>    &                                       {loogika
>>    ^
>>    |
>>    &&
>>    ||                                      }
```

Võrdlemisoperaatorid

- Võrdsus == ja mittevõrdsus !=
  - kas olulisem või mitteolulisem loogikatehetest?
  - C tegijad otsustasid, et olulisem; põhjendada võib näiteks sellega, et loogikatehete tulemusi on nii lihtsam võrrelda
  - a == b && a != c
- Võrratused >= ja <= ja võrdlused < ja > lähevad tase kõrgemale
  - kuidagi intuitiivselt on võrdsuse kontroll pigem loogika ja võrdlemine pigem aritmeetika. sellest lähtuvalt teeme aritmeetilisema poole varem ja loogilisema hiljem

Vahekokkuvõte (3)

```
>>    ()   []  ->  .                          {eriti olulised}
<<    !  ~  ++  -- + - * & (type) sizeof  {unaarsed}
>>    *  /  %                                 {aritmeetika
>>    +  -                                    }

>>    <  <=  >  >=                            {võrdlemine}
>>    == !=                                   {võrdsus}
>>    &                                       {loogika
>>    ^
>>    |
>>    &&
>>    ||                                      }
```

Omistamine

- Kuskile on vaja toppida ka omistamistehted =
  - omistada soovime üldiselt viimases hädas
  - omistame paremalt vasakule: enne arvutame parema poole valmis, siis omistame selle vasakule poolele
- Nimekiri: =, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=
- Üle jäi veel tingimuslik operaator ? :
  - tahame teha pärast kõiki tehteid, aga enne omistamist
- Tegelikult jäi üle ka , operaator
  - paneme selle kõige lõppu
  - (koma grupeerib hulka tehteid, väärtus on kõige parempoolsem komponent)

Lõplik tabel (uuesti)

```
>>    ()    []   ->   .
<<    !   ~   ++   -- + - * & (type) sizeof
>>    *   /   %
>>    +   -
>>    <<   >>
>>    <   <=   >   >=
>>    ==   !=
>>    &
>>    ^
>>    |
>>    &&
>>    ||
<<    ? :
<<    =   +=   -=   *=   /=   %=   &=   ^=   |=   <<=   >>=
>>    ,
```

chroot()

- Vahetab juurkataloogi etteantud kataloogiks

```
#include <unistd.h>
int chroot (const char *path)
```

- Kasutatakse programmide vangistamiseks etteantud kataloogi
- Käivitada tohib ainult juurkasutaja (tegelikult CAP_SYS_CHROOT protsesside puhul)
- Süsteemi juurkataloog muudetakse protsessi jaoks ära
- / algav *path* hakkab näitama vastavasse kataloogi nagu ka antud kataloogi .. (**NB! Ei midagi rohkem**)

chroot() kasutamine

- Mida jälgida?
  - chroot() nõuab protsessilt vastavat privileegi
  - pärast chrooti tuleb jälgida, et protsessil poleks väljaspool "vanglasüsteemi" avatud faile (fchdir käsu oht)
  - töökataloog tuleb muuta chrooditavaks kataloogiks ise
  - hea oleks ka kasutaja setuid()-ga ära muuta (et väljamurdmiseks uuesti chroot'i ei saaks teha)
  - teiste programmide ja librade antud keskkonnas kasutamiseks on vaja nad sellesse sisse tuua

assert()

- Assert on makro, mis kontrollib debugimisel võimalikke sisemisi vastuolusid, et mingi väärtus erineks nullist ja katkestab töö, kui see ei kehti

```
#include <assert.h>
void assert(int condition);
```

- Asserti väljalülitamiseks tuleb enne assert.h include'i defineerida NDEBUG
- Aga kui see programmi ei aeglusta, ei ole paha lisakontrolle omada

```
file:linenum: function: Assertion `expression' failed.
```

assert() (2)

- Vea korral antakse info kus viga esines, ning kutsutakse välja abort()
- abort() funktsioon tekitab signaali SIGABRT, mis põhjustab programmi töö lõppemise (aga võid ka *handler*i kirjutada)

Paroolide küsimine

- Probleem: kasutajalt parooli küsimine nii, et seda ei näidataks ekraanile (kui ta tahvlil praktikumi seletab näiteks)
- Lahendus:

```
#include <unistd.h>
char * = getpasswd(char * prompt)
```

- Võtab ühenduse "päris" terminaliga /dev/tty , kui ei õnnestu, võtab appi stdin ja stderr . Blokeeritakse INTR, QUIT ja SUSP tähemärgid terminalis.
- Terminal flushitakse enne ja pärast parooli kirjutamist

Mida getpass() teeb?

- Prindib välja argumendi prompt ja tagastab pointeri paroolistringile
- Paneb terminali režiimi, kus tähti välja ei näidata ja taastab esialgse olukorra pärast parooli
- Kuna ei ole *thread-safe*, ning POSIX standardist välja jäetud, ei soovitata kasutada ja võiks pigem ise kirjutada
- Oluliste programmide puhul on hea tava parool kiiresti krüptida ja koht, kus tähed mälus asusid esimesel võimalusel nullidega üle kirjutada.

Krüptimine: crypt()

- Krüpteerib parooli DES või MD5 algoritmiga:

```
char * crypt(constchar* key, const char* salt);
```

- Usutakse, et kui funktsiooni väljund on antud, siis parim viis algne parool leida, on võimalikud variandid ükshaaval läbi proovida.
  - Õigemini: DES algoritmi puhul ei tasu seda üldse uskuda, MD5 puhul natuke võib
- salt: kui on kahetäheline, valib DES algoritmi, kui soovid MD5, alusta stringi 1 + kuni 8 tähte, mis lõppevad $ või \0 märgiga

Salt

- Parooli sissesoolamine on tarvilik selleks, et pelgalt parooli räsi omamisest ei piisaks: ründaja peaks omama parooli räsi kõikide võimalike soolamiste puhuks.
- Väljund on salt + $ (kui algselt polnud) + räsi
- Parooli määramisel panna salt mingiks piisavalt juhuslikuks stringiks.
- Parooli kontrollimisel anda eelmine crypt() väljund ette salt argumendina, ning võrrelda *salt* ning crypt() tulemus omavahel. (kuna $ lõpetab, siis võib anda kogu tulemuse)

Keywordid

- auto break case char const continue default do double else enum extern float for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while

Register

- Teatab kompilaatorile, et optimeerimiseks võiks antud muutuja panna registrisse
- Kompilaator võib käsku ignoreerida
- Enamasti saab optimeerimisel ise aru, mida protsessoriregistris hoida
- register int counter;

Volatile

- Samuti andmetüübi kohta. Volatile muutujaid ei püüta optimeerida.
- Tähendab, et sisu võib suvalisel ajahetkel muutuda, seda kas mõne *handler*i või välise mõju tõttu. Idee on selles, et programm peaks kasutama alati muutuja füüsilist aadressi ja andmeid mitte puhverdama

```
#define TTYPORT 0x17755l
volatile char *port17 = (char *)TTYPORT;

*port17 = 'o';  /* see rida "optimeeruks" */
*port17 = 'N';
```

Mitteametlikke keyworde

- Ei ole porditavad igale poole
- fortran, pascal
  - fortran int(*fun)(int x);

- asm – assemblerkoodi lisamiseks

```
asm {
      mov EDX, port[EBP]
      mov AL, x[EBP]
      out DX, AL
}
```

Uue informatsiooni lõpp

- Järgmisel tunnil kordamine ja eksaminäide